Analisis Fungsi Hash pada Java DigestUtils

Johanes and 13517012[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13517012@std.stei.itb.ac.id

***Abstrak*—Fungsi *hash* telah digunakan di banyak aplikasi yang kita gunakan, sehingga kinerja setiap fungsi menjadi hal yang perlu diperhatikan melihat banyaknya fungsi *hash* yang ada. Salah satu *library* yang masih banyak digunakan terkait fungsi *hash* ini adalah *library* codec khususnya pada class DigestUtils yang mempunyai banyak fungsi *hash*. Pada makalah ini akan dilakukan pengujian terkait waktu eksekusi pada semua fungsi *hash* yang ada pada class DigestUtils tersebut. Pengujian akan dilakukan dengan lima kasus uji dengan jumlah kata yang berbeda-beda. Dari hasil pengujian didapatkan algoritma SHA-2-512 memiliki waktu eksekusi yang paling cepat dan algoritma MD2 memiliki waktu eksekusi yang paling lambat. Selain itu algoritma lainnya memiliki waktu eksekusi yang sangat mirip sehingga dapat digunakan secara bergantian dengan tetap memperhatikan ukuran *message digest* dan keamanan terkait *collision*.**

Kata Kunci—collision,* DigestUtils, fungsi *hash*, *message digest

I. Pendahuluan

Fungsi *hash* adalah suatu fungsi yang didesain untuk memetakan masukan yang berupa *byte* dengan ukuran sembarang menjadi suatu *byte* yang unik dengan panjang yang ditentukan. Fungsi *hash* didesain sebagai fungsi yang ireversibel sehingga hasil *hash*-nya tidak dapat ditransformasi menjadi masukan semula.

Pada umumnya fungsi *hash* banyak digunakan pada bidang kriptografi namun banyak juga implementasi fungsi *hash* pada aplikasi yang kita gunakan sehari-hari, misalnya *indexing* pada basis data, pada protokol keamanan digital, maupun pada beberapa struktur data untuk mempercepat kinerja aplikasi. Untuk sebab itu dibutuhkan fungsi hash yang sesuai dengan kebutuhan yang ada.

Umumnya yang dipertimbangkan dalam memilih fungsi *hash* adalah panjangnya *message digest* yang dihasilkan dan waktu eksekusinya. Semakin pendek *digest*-nya, maka kemungkinan *hash* yang dihasilkan akan semakin terbatas, namun semakin panjang *digest*-nya maka dibutuhkan komputasi yang lebih banyak. Telah banyak dilakukan iterasi untuk meningkatkan kinerja dari fungsi *hash* yang sudah ada mulai dari SHA-1 hingga SHA-3 dan ada pula fungsi *hash* lain seperti MD.

Salah satu *library* fungsi *hash* yang banyak digunakan adalah library codec dari apache commons. Berdasarkan data dari Maven, terdapat lebih dari 3000 aplikasi yang menggunakan *library* ini dalam 2 tahun terakhir ini [1]. Salah satu fungsi yang terdapat pada library ini adalah fungsi yang *hash* yang terdapat pada class digest.DigestUtils. Pada makalah ini akan dilakukan pengujian kinerja dari setiap fungsi *hash* yang terdapat pada DigestUtils untuk mengetahui waktu eksekusinya saat melakukan *hash* pada kata acak.

II. Dasar Teori

[2] Fungsi *hash* adalah fungsi yang mengkompresi pesan dengan ukuran sembarang menjadi *string* yang berukuran *fixed*. Hasil keluaran dari fungsi *hash* umumnya disebut *message digest* atau *hash value*. Fungsi *hash* bersifat ireversibel sehingga pesan yang sudah di-*hash* tidak dapat ditransformasi kembali menjadi pesan semula. Beberapa sifat dari fungsi *hash* adalah sebagai berikut:

1. *collision resistance*, yang berarti sulit untuk menemukan dua buah pesan yang memiliki *message digest* yang sama.
2. *preimage resistance*, yang berarti sukar untuk menemukan pesan yang memiliki nilai *hash* sesuai yang diinginkan.
3. *second preimage resistance*, yang berarti sukar menemukan suatu pesan yang memiliki nilai *hash* yang sama dengan suatu pesan lainnya.

Dengan sifat-sifat tersebut, terdapat beberapa aplikasi dari fungsi *hash* ini, yaitu:

1. Menjaga integritas pesan
   Fungsi *hash* dapat menjaga integritas pesan karena setiap perubahan pada pesan akan mengakibatkan perubahan yang signifikan terhadap nilai *hash*-nya, sehingga tidak mudah untuk melakukan modifikasi pada pesan.
2. Menghemat waktu pengiriman
   Dengan menggunakan fungsi *hash*, proses pengiriman pesan sebagai media verifikasi dapat dipersingkat karena hanya perlu mengirimkan nilai *hash*-nya saja dibandingkan harus mengirimkan seluruh pesannya.
3. Menormalkan panjang data yang beragam
   Karena nilai *hash* yang panjangnya tetap, maka setiap pesan dapat dinormalkan sehingga dapat memiliki panjang yang sama.

Berikut ini adalah beberapa contoh fungsi *hash* yang terkenal dan ada pada DigestUtils.

A. Secure Hash Algorithm (SHA)
   [3] SHA dibuat oleh NIST dan digunakan sebagai standar fungsi *hash*. Algoritma SHA dibuat berdasarkan algoritma MD4 yang dibuat oleh Ronald L. Rivest. Berikut ini adalah beberapa jenis algoritma SHA yang ada pada DigestUtils.

1. SHA-1
   SHA-1 pertama kali dipublikasikan pada tahun 1995 [4] yang akan menghasilkan keluaran berukuran 160 bit. Algoritma SHA-1 ini telah ditemukan *collision*-ya sehingga tidak aman digunakan
2. SHA-2
   SHA-2 dibuat karena telah ditemukannya *collision* pada SHA-1. Terdapat dua macam algoritma untuk SHA-2 [5] yaitu SHA-256 dan SHA-512. SHA-256 akan menghasil keluaran berukuran 256 bit yang dapat dipotong menjadi 224 bit sebagai ukuran keluaran dari SHA-224 dan SHA-512 akan menghasilkan keluaran berukuran 512 bit dan dapat dipotong menjadi 384 bit sebagai ukuran keluaran dari SHA-384. Algoritma Fungsi *hash* SHA-512 dikembangkan lebih lanjut menjadi dua buah varian yang berbeda ukuran *message digest*-nya, yaitu SHA-512-224 dan SHA-512-256 [6]. Sejauh ini belum ditemukan *collision* pada SHA-2.
3. SHA-3
   SHA-3 adalah nama dari algoritma yang memenangkan kompetisi menciptakan fungsi *hash* baru dengan tujuan untuk menggantikan fungsi SHA-1 dan fungsi SHA-2 jika telah ditemukan kolisi. Pemenang dari kompetisi tersebut adalah algoritma Keccak yang dibuat oleh Praveen Gauravaram, Lars Knudsen, Krystian Matusiewicz, Florian Mendel, Christian Rechberger, Martin Schläffer, and Søren S [7]. Terdapat empat varian dari SHA-3 ini yaitu SHA-3-224, SHA-3-256, SHA-3-284, dan SHA-3-512. Masing-masing varian memiliki proses yang berbeda pada algoritma Keccak dan memiliki panjang keluaran sesuai dengan nama variannya [8]. Hingga saat ini belum ditemukan *collision* pada algoritma SHA-3.

B. Message Digest (MD) Algorithm
   Merupakan algoritma *hash* yang dikembangkan oleh Ronald L. Rivest. Pada DigestUtils hanya terdapat dua buah varian dari MD yaitu MD2 dan MD5. Meskipun kedua fungsi *hash* ini telah ditemukan *collision*-nya, namun kedua algoritma ini masih tetap digunakan.

III. Perancangan

Fungsi *hash* yang diuji pada makalah ini adalah seluruh fungsi *hash* yang dimiliki oleh DigestUtils sebagai fungsi statik yang menerima masukan berupa *string* dan akan menghasilkan *message digest* berupa *string hex* dan fungsi tersebut belum *deprecated*. Fungsi *hash* tersebut adalah MD2. MD5, SHA-1, SHA-256, SHA-384, SHA-512, SHA-512, SHA-512-224, SHA-512-256, SHA-3-224, SHA-3-256, SHA-3-384, dan SHA-3-512.

Lingkungan pengembangan yang digunakan untuk melakukan percobaan ini adalah sebagai berikut:

- Operating System: Ubuntu 20.04
- Processor: Quad-core Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz
- RAM: 16 GiB

Pengujian dilakukan pada bahasa pemrograman Java dengan menggunakan *library* DigestUtils. Program dibuat dengan bantuan Intellij IDEA untuk mengatur konfigurasi Maven. percobaan akan dilakukan dengan mengukur waktu *hashing* setiap fungsi *hash* pada beberapa kata yang di-*generate* secara acak.

Proses pengujian akan dibagi menjadi lima kasus uji, yaitu pengujian pada kasus uji 1.000 kata acak, 10.000 kata acak, 100.000 kata acak, 1.000.000 kata acak, dan 10.000.000 kata acak. Nilai 1.000 dipilih karena untuk nilai dibawah 1.000, hasil waktu eksekusinya tidak signifikan dan di bawah orde milisekon sehingga sulit untuk dihitung sedangkan untuk nilai diatas 10.000.000 sudah terlalu lama dan lima kasus uji dirasa cukup. Setiap kasus uji akan diujikan sebanyak lima kali dan akan diambil nilai rata-ratanya. Terakhir, untuk mengurutkan waktu eksekusi, setiap rata-rata waktu akan dibagi dengan jumlah katanya sehingga didapatkan waktu estimasi rata-rata waktu eksekusi perkata. Kemudian nilai tersebut akan dirata-rata untuk setiap fungsi dan akan urutkan dari yang tercepat.

IV. Implementasi

Untuk pembangkitan kata acak akan digunakan bantuan library commons-text milik Apache dengan menggunakan class RandomStringGenerator. Akan dibangkitkan kata dengan panjang 100 karakter dengan setiap karakter merupakan karakter ASCII dengan nilai diantara desimal 32 hingga 126 karena hanya pada rentang tersebut karakter ASCII dapat di-*print*. Berikut ini adalah fungsi untuk membangkitkan kata acak tersebut.

```
static List<String> generateRandomStringList(int size) {
        List<String> randomStringList = new ArrayList<>();
            RandomStringGenerator generator = new
RandomStringGenerator.Builder().withinRange(32, 126).build();
        for (int i = 0; i < size; i++) {
            randomStringList.add(generator.generate(100));
        }
        return randomStringList;
    }
```

Kode 1. Kode program untuk membangkitkan kumpulan kata acak berdasarkan banyaknya *size*.

Variabel *size* berguna untuk menentukan banyaknya kata yang akan dihasilkan. Dalam percobaan ini banyaknya jumlah kata adalah 1.000, 10.000, 100.000, 1.000.000, dan 10.000.000 kata.

Berikut ini adalah fungsi yang digunakan untuk melakukan pengujian pada setiap fungsi *hash*.

```
static void hashTest(Object mainObject, List<List<String>>
listOfRandomStringList, Method method)
            throws InvocationTargetException,
IllegalAccessException {
        System.out.print(method.getName());
        long[] timerArray = new
long[listOfRandomStringList.size()];
        for (int i = 0; i < listOfRandomStringList.size(); i++)
{
            timerArray[i] = System.currentTimeMillis();
            for (String randomString :
listOfRandomStringList.get(i)) {
                method.invoke(mainObject, randomString);
            }
            timerArray[i] = System.currentTimeMillis() -
timerArray[i];
            System.out.print("," + timerArray[i]);

        }
        System.out.println();
    }
```

Kode 2. Fungsi untuk pengujian terhadap setiap fungsi *hash*.

Fungsi ini menerima *mainObject*, sebagai variabel yang menyimpan kelas pemanggil fungsi *hash*, *listOfRandomStringList* yang merupakan kumpulan kasus uji untuk seluruh jumlah kata acak, dan *method* yang merupakan fungsi *hash* yang akan digunakan. Waktu eksekusi setiap kasus uji akan disimpan pada *timerArray* yang merupakan *array of long* sehingga dapat menyimpan waktu dalam orde milisekon. Setelah selesai menjalankan satu kasus uji, akan ditampilkan lama waktu eksekusi ke layar.

Berikut ini adalah kode program utama pada percobaan ini.

```
public static void main(String[] args) throws
NoSuchMethodException, InvocationTargetException,
IllegalAccessException {
        List<List<String>> listOfRandomStringList = new
ArrayList<>();
        for (int size = 1000; size <= 10000000; size *= 10) {
            System.out.print(size);

listOfRandomStringList.add(generateRandomStringList(size));
            System.out.print(" done\n");
        }
        System.out.println();

        System.out.println("initiate");
        Method md2Hex = DigestUtils.class.getMethod("md2Hex",
String.class);
        Method md5Hex = DigestUtils.class.getMethod("md5Hex",
String.class);
        Method sha1Hex = DigestUtils.class.getMethod("sha1Hex",
String.class);
        Method sha256Hex =
DigestUtils.class.getMethod("sha256Hex", String.class);
        Method sha384Hex =
DigestUtils.class.getMethod("sha384Hex", String.class);
        Method sha512Hex =
DigestUtils.class.getMethod("sha512Hex", String.class);
        Method sha512_224Hex =
DigestUtils.class.getMethod("sha512_224Hex", String.class);
        Method sha512_256Hex =
DigestUtils.class.getMethod("sha512_256Hex", String.class);
        Method sha3_224Hex =
DigestUtils.class.getMethod("sha3_224Hex", String.class);
        Method sha3_256Hex =
DigestUtils.class.getMethod("sha3_256Hex", String.class);
        Method sha3_384Hex =
DigestUtils.class.getMethod("sha3_384Hex", String.class);
        Method sha3_512Hex =
DigestUtils.class.getMethod("sha3_512Hex", String.class);
        Main mainObject = new Main();

        System.out.println("run\n");
        hashTest(mainObject, listOfRandomStringList, md2Hex);
        hashTest(mainObject, listOfRandomStringList, md5Hex);
        hashTest(mainObject, listOfRandomStringList, sha1Hex);
        hashTest(mainObject, listOfRandomStringList,
sha256Hex);
        hashTest(mainObject, listOfRandomStringList,
sha3_224Hex);
        hashTest(mainObject, listOfRandomStringList,
sha3_256Hex);
        hashTest(mainObject, listOfRandomStringList,
sha3_384Hex);
        hashTest(mainObject, listOfRandomStringList,
sha3_512Hex);
        hashTest(mainObject, listOfRandomStringList,
sha384Hex);
        hashTest(mainObject, listOfRandomStringList,
sha512_224Hex);
        hashTest(mainObject, listOfRandomStringList,
sha512_256Hex);
        hashTest(mainObject, listOfRandomStringList,
sha512Hex);
    }
```

Kode 3. Kode utama untuk melakukan pengujian terhadap semua fungsi *hash* pada DigestUtils.

Pada awalnya kode ini akan membangkitkan kumpulan kata acak untuk setiap kasus uji dan akan disimpan pada variabel *listOfRandomStringList*. Kemudian akan diambil semua fungsi *hash* yang ada pada DigestUtils sehingga dapat di-*passing* ke dalam fungsi *hashTest*. Terakhir, akan dipanggil fungsi *hashTest* dengan menggunakan parameter *mainObject* sebagai objek pemanggil, *listOfRandomStringList* yang merupakan kasus uji, dan setiap fungsi *hash* yang akan diujikan.

V. Hasil Pengujian

Pengujian akan dilakukan lima kali untuk masing-masing kasus uji dan akan diambil waktu rata-rata-nya untuk diurutkan.

A. Kasus uji 1.000 kata acak

Berikut ini adalah hasil pengujian pada fungsi *hash* untuk 1.000 kata acak.

Tabel 1. Waktu pengujian pada kasus uji 1

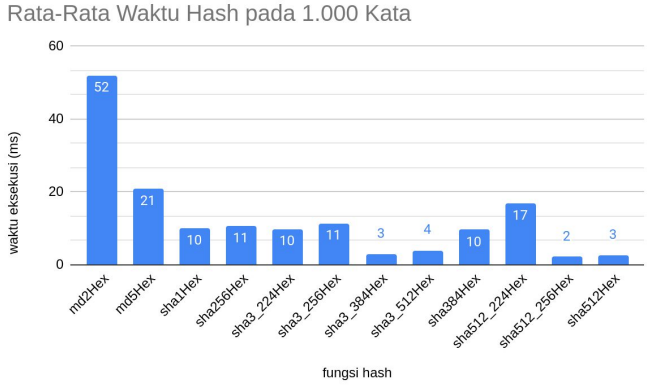
Fungsi hash	waktu 1 (ms)	waktu 2 (ms)	waktu 3 (ms)	waktu 4 (ms)	waktu 5 (ms)
md2Hex	47	49	58	55	51
md5Hex	25	21	19	19	20
sha1Hex	10	8	9	10	13
sha256Hex	11	9	10	11	12
sha3_224Hex	9	8	11	10	11
sha3_256Hex	13	9	9	13	13
sha3_384Hex	3	3	3	3	3

sha3_512Hex	4	3	3	5	5
sha384Hex	9	10	10	8	11
sha512_224Hex	22	14	20	13	15
sha512_256Hex	2	3	2	2	2
sha512Hex	3	2	2	3	3

Berikut ini adalah hasil perhitungan rata-rata dan pengurutan rata-rata waktu dari yang paling cepat.

Tabel 2. Rata-rata waktu pada kasus uji 1

Fungsi hash	rata-rata waktu (ms)
sha512_256Hex	2.2
sha512Hex	2.6
sha3_384Hex	3.0
sha3_512Hex	4.0
sha384Hex	9.6
sha3_224Hex	9.8
sha1Hex	10.0
sha256Hex	10.6
sha3_256Hex	11.4
sha512_224Hex	16.8
md5Hex	20.8
md2Hex	52.0



Grafik 1. Rata-rata waktu *hash* pada kasus uji 1

B. Kasus uji 10.000 kata acak

Berikut ini adalah hasil pengujian pada fungsi *hash* untuk 10.000 kata acak.

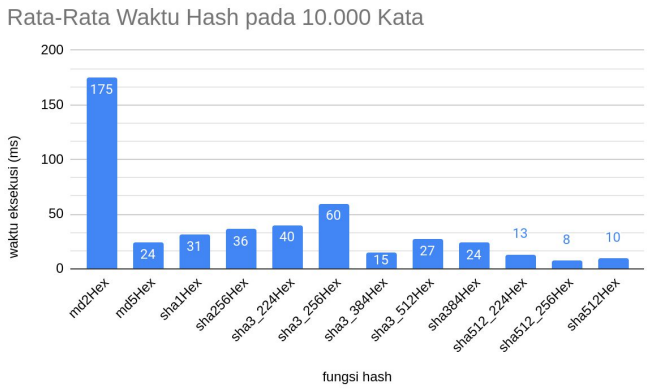
Tabel 3. Waktu pengujian pada kasus uji 2

Fungsi hash	waktu 1 (ms)	waktu 2 (ms)	waktu 3 (ms)	waktu 4 (ms)	waktu 5 (ms)
md2Hex	171	170	176	177	180
md5Hex	26	29	28	18	20
sha1Hex	43	23	42	21	27
sha256Hex	35	30	34	35	47
sha3_224Hex	36	51	43	36	34
sha3_256Hex	55	60	60	63	60
sha3_384Hex	19	12	13	19	13
sha3_512Hex	26	22	24	40	23
sha384Hex	19	20	33	23	24
sha512_224Hex	17	13	13	9	14
sha512_256Hex	8	8	8	7	8
sha512Hex	10	7	10	10	12

Berikut ini adalah hasil perhitungan rata-rata dan pengurutan rata-rata waktu dari yang paling cepat.

Tabel 4. Rata-rata waktu pada kasus uji 2

Fungsi hash	rata-rata waktu (ms)
sha512_256Hex	7.8
sha512Hex	9.8
sha512_224Hex	13.2
sha3_384Hex	15.2
sha384Hex	23.8
md5Hex	24.2
sha3_512Hex	27.0
sha1Hex	31.2
sha256Hex	36.2
sha3_224Hex	40.0
sha3_256Hex	59.6
md2Hex	174.8



Grafik 2. Rata-rata waktu *hash* pada kasus uji 2

C. Kasus uji 100.000 kata acak
Berikut ini adalah hasil pengujian pada fungsi *hash* untuk 100.000 kata acak.

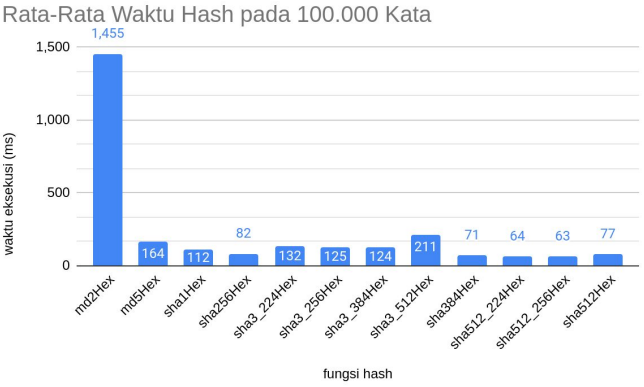
Tabel 5. Waktu pengujian pada kasus uji 3

Fungsi hash	waktu 1 (ms)	waktu 2 (ms)	waktu 3 (ms)	waktu 4 (ms)	waktu 5 (ms)
md2Hex	1,364	1,402	1,392	1,435	1,682
md5Hex	184	177	153	157	148
sha1Hex	90	114	90	113	153
sha256Hex	82	74	79	85	92
sha3_224Hex	148	133	125	127	127
sha3_256Hex	128	120	148	113	117
sha3_384Hex	129	117	118	141	113
sha3_512Hex	224	202	207	218	206
sha384Hex	75	68	70	71	71
sha512_224Hex	75	59	59	62	63
sha512_256Hex	64	65	57	67	60
sha512Hex	69	85	66	72	92

Berikut ini adalah hasil perhitungan rata-rata dan pengurutan rata-rata waktu dari yang paling cepat.

Tabel 6. Rata-rata waktu pada kasus uji 3

Fungsi hash	rata-rata waktu (ms)
sha512_256Hex	62.6
sha512_224Hex	63.6
sha384Hex	71.0
sha512Hex	76.8
sha256Hex	82.4
sha1Hex	112.0
sha3_384Hex	123.6
sha3_256Hex	125.2
sha3_224Hex	132.0
md5Hex	163.8
sha3_512Hex	211.4
md2Hex	1,455.0



Grafik 3. Rata-rata waktu *hash* pada kasus uji 3

D. Kasus uji 1.000.000 kata acak
Berikut ini adalah hasil pengujian pada fungsi *hash* untuk 1.000.000 kata acak.

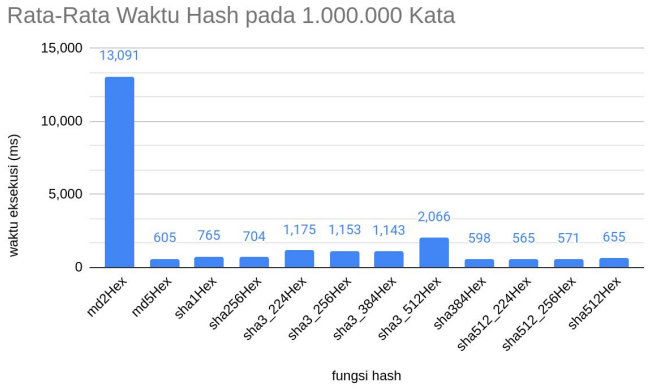
Tabel 7. Waktu pengujian pada kasus uji 4

Fungsi hash	waktu 1 (ms)	waktu 2 (ms)	waktu 3 (ms)	waktu 4 (ms)	waktu 5 (ms)
md2Hex	13,384	12,904	12,888	12,847	13,431
md5Hex	607	589	617	614	597
sha1Hex	798	722	715	722	869
sha256Hex	725	673	651	659	813
sha3_224Hex	1,258	1,190	1,114	1,117	1,194
sha3_256Hex	1,213	1,126	1,177	1,131	1,118
sha3_384Hex	1,192	1,124	1,161	1,118	1,120
sha3_512Hex	2,131	2,015	2,070	2,105	2,010
sha384Hex	632	560	569	605	623
sha512_224Hex	603	551	516	555	601
sha512_256Hex	612	555	525	608	554
sha512Hex	640	627	575	656	778

Berikut ini adalah hasil perhitungan rata-rata dan pengurutan rata-rata waktu dari yang paling cepat.

Tabel 8. Rata-rata waktu pada kasus uji 4

Fungsi hash	rata-rata waktu (ms)
sha512_224Hex	565.2
sha512_256Hex	570.8
sha384Hex	597.8
md5Hex	604.8
sha512Hex	655.2
sha256Hex	704.2
sha1Hex	765.2
sha3_384Hex	1,143.0
sha3_256Hex	1,153.0
sha3_224Hex	1,174.6
sha3_512Hex	2,066.2
md2Hex	13,090.8



Grafik 4. Rata-rata waktu *hash* pada kasus uji 4

E. Kasus uji 10.000.000 kata acak

Berikut ini adalah hasil pengujian pada fungsi *hash* untuk 10.000.000 kata acak.

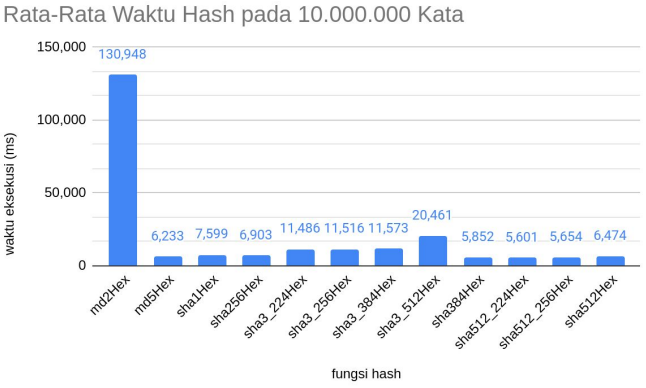
Tabel 9. Waktu pengujian pada kasus uji 5

Fungsi hash	waktu 1 (ms)	waktu 2 (ms)	waktu 3 (ms)	waktu 4 (ms)	waktu 5 (ms)
md2Hex	135,924	126,753	126,631	126,244	139,190
md5Hex	6,373	6,002	6,061	6,202	6,526
sha1Hex	7,747	7,175	7,121	7,325	8,625
sha256Hex	7,367	6,587	6,569	6,550	7,440
sha3_224Hex	12,010	11,585	11,010	11,075	11,751
sha3_256Hex	12,084	11,253	11,567	11,282	11,395
sha3_384Hex	12,003	11,388	11,591	11,538	11,344
sha3_512Hex	20,886	20,030	20,587	20,360	20,441
sha384Hex	6,085	5,656	5,606	5,748	6,166
sha512_224Hex	6,082	5,557	5,185	5,501	5,681
sha512_256Hex	5,939	5,492	5,217	5,703	5,919
sha512Hex	6,496	6,027	5,916	6,537	7,395

Berikut ini adalah hasil perhitungan rata-rata dan pengurutan rata-rata waktu dari yang paling cepat.

Tabel 10. Rata-rata waktu pada kasus uji 5

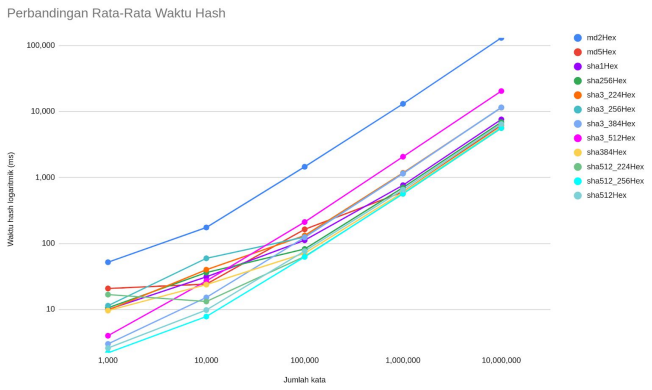
Fungsi hash	rata-rata waktu (ms)
sha512_224Hex	5,601.2
sha512_256Hex	5,654.0
sha384Hex	5,852.2
md5Hex	6,232.8
sha512Hex	6,474.2
sha256Hex	6,902.6
sha1Hex	7,598.6
sha3_224Hex	11,486.2
sha3_256Hex	11,516.2
sha3_384Hex	11,572.8
sha3_512Hex	20,460.8
md2Hex	130,948.4



Grafik 5. Rata-rata waktu *hash* pada kasus uji 5

VI. Analisis

Kemudian hasil pengujian tersebut digabungkan dan dipetakan pada skala logaritmik. Berikut ini adalah hasil pemetaan tersebut.



Grafik 6. Grafik pemetaan logaritmik antara jumlah kata dan waktu eksekusi

Untuk grafik lebih jelas dapat dilihat pada bagian lampiran di bawah.

Berdasarkan grafik tersebut dapat dilihat waktu eksekusinya linier untuk jumlah kata yang besar dan terdapat sedikit *overhead* pada jumlah kata yang kecil. Oleh sebab itu tidak dilakukan pengujian untuk jumlah kata yang lebih sedikit. Berdasarkan hasil eksperimen, dilakukan perhitungan untuk menentukan urutan waktu *hashing*. Perhitungan dilakukan dengan cara, setiap rata-rata waktu eksekusi dibagi dengan jumlah katanya sehingga didapatkan estimasi rata-rata waktu eksekusi per kata. Kemudian nilai tersebut dirata-rata untuk semua kasus uji. Berikut ini adalah hasil perhitungan tersebut yang sudah diurutkan dari waktu eksekusi yang paling cepat. Waktu yang tertera dalam mikrosekon.

Tabel 11. Estimasi rata-rata waktu eksekusi untuk setiap kata

Fungsi hash	rata-rata waktu (us)
sha512_256Hex	0.94844
sha512Hex	1.130124
sha3_384Hex	1.611256
sha3_512Hex	2.585256
sha384Hex	2.774604
sha1Hex	3.153012
sha256Hex	3.287692
sha3_224Hex	3.488644
sha512_224Hex	3.976264
sha3_256Hex	4.183324
md5Hex	5.217216
md2Hex	22.043128

Tabel 12. Estimasi rata-rata waktu eksekusi untuk setiap kata tanpa kasus uji 1

Fungsi hash	rata-rata waktu (us)
sha512_256Hex	0.63555
sha512Hex	0.762655
sha512_224Hex	0.77033
sha384Hex	1.068255
sha3_384Hex	1.26407
md5Hex	1.32152
sha1Hex	1.441265
sha256Hex	1.459615
sha3_224Hex	1.910805
sha3_512Hex	2.23157
sha3_256Hex	2.379155
md2Hex	14.55391

Tabel 11 merupakan hasil estimasi rata-rata yang mencangkup seluruh kasus uji sedangkan pada tabel 12 hanya menggunakan kasus uji 2 hingga kasus uji 5, hal ini dikarenakan adanya *overhead* pada kasus uji 1 dan berdasarkan pengamatan pada grafik 6, rata-rata waktu eksekusi cenderung lebih linear jika dilihat dari kasus uji 2 hingga kasus uji 5. Berdasarkan tabel 11 dan tabel 12, dapat dilihat bahwa algoritma *hash* yang paling cepat adalah algoritma SHA-2-512. Sedangkan algoritma yang paling lambat adalah MD2.

Berdasarkan analisis dari referensi [5][6][8] seharusnya algoritma SHA-1 lebih cepat daripada SHA-2 dan SHA-2 lebih cepat dari pada SHA-3. Hal tersebut dikarenakan kompleksitas algoritma yang lebih tinggi dan keamanan yang lebih pada SHA-3. Namun pada percobaan ini algoritma SHA-1 lebih lambat daripada SHA-2 dan SHA-3. Alasan mengapa hal tersebut terjadi masih belum diketahui secara pasti. Beberapa kemungkinannya adalah, implementasi dari fungsi tersebut kurang optimal dibandingkan dengan kedua fungsi lainnya atau terjadi gangguan sistem saat pengujian algoritma SHA-1 misalnya adanya proses lain yang berjalan atau dikarenakan suatu proses pada JVM (Java Virtual Machine).

VII. Kesimpulan

Percobaan ini tidak melihat panjangnya *message digest* yang dihasilkan dan hanya mengamati waktu eksekusinya saja. Berdasarkan waktu eksekusi dan ukuran *message digest*-nya dapat dipilih algoritma yang cocok untuk digunakan. Jika hanya membutuhkan waktu yang singkat dapat digunakan algoritma *hash* SHA-2-512 dan dipilih ukuran *message digest*-nya antara 512 bit, 224 bit, 384 bit, atau 256 bit. Sedangkan jika ingin algoritma yang aman lebih aman dapat digunakan fungsi SHA-3 dan dapat dipilih ukuran *message digest* yang dibutuhkan.

Selain mencari yang tercepat, mungkin saja digunakan

fungsi *hash* yang lambat dengan tujuan untuk menghambat proses *brute force attack*. Salah satu aplikasi yang mungkin adalah pada pengamanan *password* yang menggunakan fungsi *hash*. Dengan waktu eksekusi yang lebih lambat, dibutuhkan waktu total yang lebih lama untuk mencari *password* yang benar yang dapat membantu mencegah serangan terhadap *password* yang disimpan.

Namun berdasarkan grafik 6, secara umum fungsi *hash* memiliki hasil waktu eksekusi yang hampir sama. Beberapa fungsi yang perlu diperhatikan adalah fungsi MD2 dan SHA-3-512 yang terlihat jelas memiliki waktu eksekusi yang lambat. Selain kedua fungsi tersebut, fungsi *hash* lainnya dapat digunakan secara bergantian, hanya perlu diperhatikan ukuran *message digest* dan keamanannya terkait *collision*.

VIII. Ucapan Terima Kasih

Terima kasih kepada Tuhan yang Maha Esa, yang oleh karena berkat dan rahmat-Nya makalah pengganti UAS ini dapat diselesaikan. Terima kasih juga diucapkan kepada Dr. Ir. Rinaldi Munir, M.T. sebagai dosen pengajar karena sudah mengajarkan sangat banyak hal mengenai kriptografi pada semester ini. Terakhir, terima kasih juga disampaikan kepada semua teman dan keluarga yang membantu dalam menyelesaikan makalah ini.

Referensi

[1] "Maven Repository: commons-codec » commons-codec." https://mvnrepository.com/artifact/commons-codec/commons-codec (accessed Dec. 20, 2020).

[2] R. Munir, "Fungsi Hash." http://informatika.stei.itb.ac.id/~rinaldi.munir/Kriptografi/2020-2021/Fungsi-hash-2020.pdf (accessed Dec. 20, 2020).

[3] R. Munir, "Secure Hash Algorithm (SHA)." http://informatika.stei.itb.ac.id/~rinaldi.munir/Kriptografi/2020-2021/Fungsi-hash-SHA.pdf (accessed Dec. 20, 2020).

[4] "SHA-1 - Wikipedia." https://en.wikipedia.org/wiki/SHA-1 (accessed Dec. 20, 2020).

[5] "The attached publication, FIPS Publication 180-2 (with Change Notice 1) (change notice dated," 2004. Accessed: Dec. 20, 2020. [Online]. Available: http://csrc.nist.gov/publications/PubsFIPS.html#fips180-4.

[6] W. E. May, "FIPS PUB 180-4 FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION Secure Hash Standard (SHS) CATEGORY: COMPUTER SECURITY SUBCATEGORY: CRYPTOGRAPHY," 2012, doi: 10.6028/NIST.FIPS.180-4.

[7] R. Munir, "SHA-3 (Keccak)." http://informatika.stei.itb.ac.id/~rinaldi.munir/Kriptografi/2020-2021/SHA-3-2020 (accessed Dec. 20, 2020).

[8] J. Foti, "FIPS PUB 202 FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions CATEGORY: COMPUTER SECURITY SUBCATEGORY: CRYPTOGRAPHY," 2015, doi: 10.6028/NIST.FIPS.202.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 20 Desember 2020

Johanes 13517012

LAMPIRAN

A. Grafik pemetaan logaritmik antara jumlah kata dan waktu eksekusi